# 060: Library Design for Parallel Computation

Most of the design considerations presented in the text book are not very well reflected in below. You need the context, in which the exercises have been posted, to understand the design story (which is definitely expected for a good grade). Read through the book text carefully while solving exercises.

All exercises are to be solved by extending the file `Par.scala`—the only file you should hand in. This exercise sheet is short, ranked for about 2 hours. However, this may be deceiving. It is essentially impossible to solve it without reading the chapter in parallel (which will take much more time).

**Exercise 1.** Use `lazyUnit` to write a function that converts any function `A =>B` to one that evaluates its result asynchronously (so it spawns a separate thread).

```
def asyncF[A,B](f: A =>B): A =>Par[B]
```

A suitable place to start is marked Exercise 1 in `Par.scala`.[1]

**Exercise 2.** Write a function `sequence` that takes a list of parallel computations (`List[Par[B]]`) and returns a parallel computation producing a list (`Par[List[B]]`). No additional primitives are required. Don't call run, as we do not want this function to execute anything yet.[2]

```
def sequence[A](ps: List[Par[A]]): Par[List[A]]
```

**Exercise 3.** Implement `parFilter`, which filters elements of a list in parallel. [3]

```
def parFilter[A](as: List[A])(f: A =>Boolean): Par[List[A]]
```

**Exercise 4.** Implement `map3` using `map2`.

```
def map3[A,B,C,D] (pa :Par[A], pb: Par[B], pc: Par[C]) (f: (A,B,C) =>D)
```

**Exercise 5.** Implement `choiceN` and then `choice` in terms of `choiceN`.[4]

**Exercise 6.** Implement a general parallel computation `chooser`, and then use it to implement `choice` and `choiceN`. A `chooser` uses a parallel computation to obtain a selector for one of the available parallel computations in the range provided by `choices`:[5]

```
def chooser[A,B](pa: Par[A])(choices: A =>Par[B]): Par[B]
```

**Note:** In search for an "aha" moment, compare the type of the chooser, with the types of `Option.flatMap`, `Stream.flatMap`, `List.flatMap` and `State.flatMap`. Observe that the `chooser` is used to compose (sequence) to parallel computations here.

**Exercise 7.** Implement `join`. Can you see how to implement `flatMap` using `join`? And can you implement `join` using `flatMap`?

```
def join[A](a: Par[Par[A]]): Par[A]
```

Compare the type of `join` with the type of `List.flatten` (and the relation of `join` to `chooser` against the relation of `List.flatten` to `List.flatMap`).

[1] Exercise 7.4 [Chiusano, Bjarnason 2014]
[2] Exercise 7.5 [Chiusano, Bjarnason 2014]
[3] Exercise 7.6 [Chiusano, Bjarnason 2014]
[4] Exercise 7.11 [Chiusano, Bjarnason 2014]
[5] Exercise 7.13 [Chiusano, Bjarnason 2014]